ARL-TR-8213 ● Nov 2017

US Army Research Laboratory

Pre- and Post-Processing Tools to Create and Characterize Particle-Based Composite Model Structures

by Michael E Fortunato, Joseph Mattson, DeCarlos E Taylor, James P Larentzos, and John K Brennan

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8213 ● Nov 2017

US Army Research Laboratory

Pre- and Post-Processing Tools to Create and Characterize Particle-Based Composite Model Structures

by DeCarlos E Taylor, James P Larentzos, and John K Brennan
Weapons and Materials Research Directorate, ARL

Michael E Fortunato
University of Florida, Department of Chemistry, Gainesville, FL

Joseph Mattson
Science and Math Academy, Aberdeen High School, Aberdeen, MD

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
November 2017	Technical Report	15 May 2017–01 September 2017

4. TITLE AND SUBTITLE
Pre- and Post-Processing Tools to Create and Characterize Particle-Based Composite Model Structures

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Michael E Fortunato, Joseph Mattson, DeCarlos E Taylor, James P Larentzos, and John K Brennan

5d. PROJECT NUMBER
HIP-17-015 (HPCMP Internship Program)

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
Weapons and Materials Research Directorate (ATTN: RDRL-WML-B)
2800 Powder Mill Road
Adelphi, MD 20783-1138

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8213

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)
HPCMP

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Microstructural heterogeneities, such as voids, cracks, and grain boundaries, dictate the macroscopic material's properties as well as its response to the thermal and mechanical loading that occurs in most applications and technologies. Simulating these materials at the microscale requires sophisticated computational tools to efficiently build and analyze the structures. This work describes a suite of computational tools developed to both create coarse-grain composite model structures and characterize their structure and material properties. The US Army Research Laboratory's Composite Model Builder and Analysis Toolkit leverages the existing parallel communication framework within the Large-scale Atomic/Molecular Massively Parallel Simulator and/or the Python mpi4py library to efficiently process systems containing billions of particles, thus enabling the study of microstructural heterogeneity in composite materials.

15. SUBJECT TERMS
Microstructure, composite materials, polymers, void detection, Composite Model Builder and Analysis Toolkit, COMBAT, python, LAMMPS, AMMo-PM

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James P Larentzos
Unclassified	Unclassified	Unclassified	UU	38	19b. TELEPHONE NUMBER (Include area code) (410) 306-0809

Contents

List of Figures

List of Tables

Acknowledgments

The authors are grateful to Shawn Coleman (US Army Research Laboratory [ARL]), Dan Foley (Drexel University), and Garritt Tucker (Colorado State University) for providing the initial version of the polycrystal builder and helping with its implementation; Tim Sirk (ARL) for discussions on building model polymer chains; Mitch Wood (Sandia National Laboratories) for suggesting the rigid body dynamics capability within the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS); and Aidan Thompson and Steve Plimpton (Sandia National Laboratories) for general guidance on utilizing LAMMPS capabilities. The authors acknowledge funding from the Department of Defense (DOD) High Performance Computing Modernization Program Internship Program (HIP-17-015) and the Office of Naval Research. This work was supported in part by a grant of computer time from the DOD High Performance Computing Modernization Program at the ARL DOD Supercomputing Resource Center.

INTENTIONALLY LEFT BLANK.

1. Introduction

Particle-based simulations at micrometer-length scales are required in order to study the effects of microstructural features that commonly exist in composite materials. These microstructural heterogeneities, such as voids, cracks, and grain boundaries, dictate the macroscopic material's properties, as well as its response to the thermal and mechanical loading that occurs in most applications and technologies. Access to these length scales can be computationally taxing for fully atomistic simulations; thus, micro- and mesoscale methods involving coarse-graining are commonly employed to extend the spatial and temporal scales. While coarse-graining reduces the number of degrees-of-freedom in the system, there remains a "big data" challenge in efficiently processing extremely large data sets containing information for *O*(billions) of particles over the entire trajectory of a simulation. This work describes a suite of computational tools developed to both create coarse-grain composite model structures, and characterize their structure and material properties. These computational tools, hereafter referred to as the ARL Composite Model Builder and Analysis Toolkit (COMBAT), leverage the existing parallel communication framework within the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)[1] and/or the Python mpi4py[2] library to efficiently process systems containing billions of particles, thus enabling the study of microstructural heterogeneity in composite materials.

The technical report is organized as follows: an overview of the underlying software used in the ARL COMBAT suite is provided in Section 2, followed by a description of pre-processing tools in Section 3, post-processing tools in Section 4, and example usage of the tool features in the Appendix. Most of the tools can be used independently, so readers that are interested in solely polymer building or void detection can seek out those subsections and the corresponding example code usage independently.

2. Software

The software tools described here leverage functionality present within the LAMMPS[1] software package, or were developed independently in a Python module, *combat*, to analyze data sets produced by LAMMPS. The following sections present a brief overview of LAMMPS and the *combat* Python module, including dependencies and capabilities. Readers are directed to the LAMMPS website at http://lammps.sandia.gov for more thorough documentation on the LAMMPS framework and user commands. The *combat* Python module is

available at https://github.com/USArmyResearchLab under the Army Materials Modeling for Particle Models software suite (pending ARL Public Release approval).

2.1 LAMMPS

LAMMPS[1] is a highly-scalable domain decomposition software developed by the Department of Energy at Sandia National Laboratories, and can be used to perform a variety of simulations from the atomistic scale to the continuum scale. LAMMPS efficiently scales in parallel, and is routinely used to perform particle-based simulations containing *O*(billions) of particles. Functionality is partitioned into separate add-on packages located in the LAMMPS source directory that can be optionally included when building/installing LAMMPS. The following add-on packages are integral to the software tools described hereafter, and must be installed in order to follow the examples provided in the Appendix:

- MC: required for polymer bond creation
- MISC: required for monomer insertion
- MOLECULE: required for polymer growth
- RIGID: required for rigid expansion of polycrystalline materials
- VORONOI: required for void volume calculations in crystalline materials

2.2 *Combat* Python Package

A Python module, *combat*, was developed in order to efficiently post-process data sets produced by LAMMPS. The module contains code to perform domain decomposition, build neighbor lists, and compute locally weighted averages in parallel. The spatial domain decomposition process involves identifying both the particles located within the domain (hereafter referred to as "local particles"), and the particles that are not located within a domain, but are within the cutoff distance of any particle within the domain (hereafter referred to as "ghost particles"). Periodic boundary conditions in 3 dimensions are considered in this process. Once the system has been subdivided, each processor is assigned an individual subdomain, containing all the necessary information to perform calculations that rely on neighboring particles (e.g., Lucy weighted averages[3]), and can execute commands independently to dramatically reduce the time-to-solution and improve parallel efficiency. Additionally, data is structured on each processor as a pandas[4] DataFrame, allowing complex querying and efficient calculations on the tabulated data. Additional details on file input/output (I/O),

domain decomposition, and neighbor list construction are discussed in the following sections.

2.2.1 Input/Output

In the present implementation of *combat*, the reading and writing of data files is currently limited to LAMMPS data file and dump file formats, but can be readily extended to other file formats with appropriate modifications. The following code is used to read a dump file format in serial, and include a buffer region or "skin" of periodic neighboring particles within a 10 Å cutoff (required for accurate neighbor computations):

```
s = combat.System.from_dump(filename, cutoff=10.0)
```

The `combat.System.from_dump` class method returns an object with attribute `particles` that is a pandas DataFrame containing per-particle data. The I/O procedure changes slightly when performing analysis in parallel, where the following code is the parallel analog using mpi4py[2]:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
s = combat.System.from_dump(filename, comm=comm, cutoff=10.0)
```

At this point in the code execution, each processor can independently perform computations locally on their own DataFrame using `s.particles`.

Note that when performing calculations within pandas, additional columns may be created within the DataFrame. By default, when attempting to use *combat* to write a new, modified output dump file, all columns that exist in the pandas DataFrame will be written to the output dump file. This may be undesirable or problematic in cases where large output dump files are produced. For instance, when a neighbor list has been created (discussed in Section 2.2.3), a new column is added to the pandas DataFrame containing the neighbors for each particle. Often, it may be unnecessary to print this information to the output dump file. In general, it is good practice to remove any undesirable data columns from the pandas DataFrame before writing to the output dump file.

As an example, some computations within *combat* may build a neighbor list stored as a column named "neighbors", while other computations may build a neighbor list stored as a column named "iloc_neighbors". These data columns correspond to lists of index values and integer locations in the DataFrame, respectively, and are solely used for efficient execution within pandas, but these data columns are not needed in the output dump file. In this case, the following checks should be included to ensure that undesired information is not written to the dump file:

```
if 'neighbors' in s.particles:
    del s.particles['neighbors']
if 'iloc_neighbors' in s.particles:
    del s.particles['iloc_neighbors']
```

Similar commands can be used to remove any other undesirable data columns contained within the pandas DataFrame.

The following code is used to write a dump file in serial:

```
s.write_dump(filename, header=True, ghost=False)
```

The header and ghost keyword arguments are optional. These keywords designate if the header information is included, and if particles identified as ghost particles are included in the dump file (ghost particles are further discussed in Section 2.2.2).

The following code can be used to write a dump file in parallel:

```
from subprocess import Popen, PIPE
import numpy as np

if comm.rank==0:
      print('gathering global number of particles')
nparticles=comm.gather(len(s.particles[s.particles['ghost']==0]),
root=0)
if comm.rank == 0:
      s.global_particles = np.sum(nparticles)
      s.dump_header('dump.header')
s.write_dump('dump.{}p'.format(comm.rank), header=False, ghost=False)
comm.Barrier()
if comm.rank == 0:
      print('concatenating')
      Popen('cat dump.header dump.*p > dump', shell=True).communicate()
      Popen('rm dump.header dump.*p', shell=True).communicate()
```

2.2.2 Spatial Domain Decomposition

Spatial domain decomposition partitions a 3-D space into smaller subdomains of equal volume, such that the assignment of *n* processors to *n* subdomains distributes computational cost across all processors. For a set of particles of uniform density in an orthorhombic simulation cell, this can be accomplished by creating a 3-D regular grid of processors that is superimposed onto the simulation cell. Each processor is responsible for the particles that reside in its subdomain (the local particles). In addition, each processor must include a buffer region or "skin" surrounding its subdomain, so as to include all neighboring particles within a cutoff distance (the ghost particles). A final level of complexity is added when considering periodic boundary conditions, which requires processors with

subdomains located at the global simulation cell boundary to track particles on the opposite end of the simulation cell.

The current implementation of *combat* uses MPI enabled by the mpi4py Python package[2]. Each processor independently reads the entire data file containing particle coordinates, and performs the following operations:

1) Define the subdomain boundaries in each dimension, and claim ownership of all particles that reside within the subdomain.

2) Identify ghost particles from neighboring subdomains that reside within a specified cutoff distance.

3) Recreate a skin of periodic images of particles for each dimension identified as periodic, and repeat the ghost particle identification in Step 2.

4) Delete all particles from memory that are not owned and are not ghost particles.

At this point in the procedure, each processor has sufficient information to calculate local properties, and can proceed independently until further communication of global properties or the output of data to the memory is required. Ultimately, in order to write a new data file with newly computed per-particle properties, each processor writes its own file containing data for the local particles it owns, and then the processor with rank 0 concatenates the files into one data file.

Figure 1 (left) shows an example spatial-domain decomposition of a sample structure using 8 processors. Each colored region represents a different subdomain for which a different processor is responsible for the particles contained within the subdomain. Using a cross-sectional view, Fig. 1 (right) shows the particles that processor 0 is responsible for monitoring, where the local particles are colored blue and the ghost particles are colored white. The domain boundaries for processor 0 are shown as black lines.

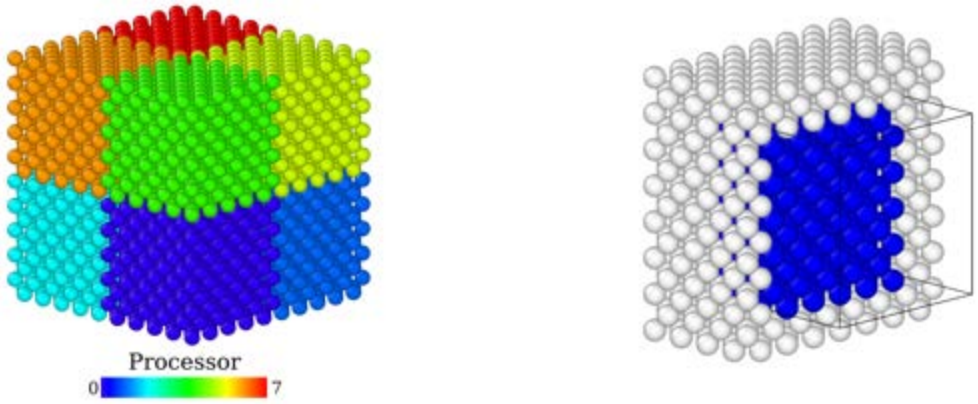

Fig. 1 (Left) An example domain decomposition with 8 processors. (Right) Cross-sectional view of the particles tracked by processor 0. Blue local particles fall within the processor's domain, while white ghost particles fall outside the domain. The processor must maintain the ghost particles because they fall within a cutoff distance from the edge of its domain.

2.2.3 Building Neighbor Lists

By minimizing the amount of data each processor contains, neighbor list construction and neighbor-dependent calculations can be performed more efficiently. Each processor uses a pandas DataFrame data structure to track its particles, and constructs neighbor lists using the SciPy[5] KDTree data structure to efficiently identify nearest neighbors. By leveraging pandas, SciPy, and NumPy[6], as well as the efficient vectorized operations contained within those Python packages, neighbor-dependent calculations can be computed orders of magnitude faster in wall-clock time as compared to pure "pythonic" algorithms.

Neighbor lists that store the index values of neighbor particles as well as the integer location in the DataFrame can be constructed. These columns are stored in the DataFrame and labeled as "neighbors" or "iloc_neighbors" accordingly.

Methods that require a neighbor list will determine if a neighbor list needs to be created/updated. Alternatively, the following can be used to manually build (or update) neighbor lists:

```
s.neighbors_all(cutoff=10.0, remove_self=True)
s.iloc_neighbors(cutoff=10.0, remove_self=True)
```

The `remove_self` keyword argument is optional, and determines whether a particle will appear in its own neighbor list. This parameter allows for control where some calculations that require neighbor lists may include self-contributions, while others may not (e.g., a local density calculation).

3. Pre-Processing Tools

3.1 Polycrystal Building

The examples contained in this technical report use a Voronoi-based nanocrystal builder Python tool developed previously at ARL.[7] The nanocrystal builder reads a LAMMPS dump file of a reference single crystal, and applies a Voronoi tessellation scheme to create a polycrystalline sample with a user-defined average grain size distribution (see Sections A.1 and A.2 of the Appendix for details of building the single crystal and polycrystalline samples, respectively). For application to the examples provided in the Appendix, it is required that each grain in the polycrystalline sample is identified using a different molecule ID within the LAMMPS data file format. Additionally, image flags must be accurate; this information is necessary for performing rigid-body dynamics. An example polycrystalline sample that was constructed using the Voronoi-based nanocrystal builder[7] is shown in Fig. 2 (left), with grain boundary regions colored white. This sample structure was built by first creating a single crystal face-centered cubic (FCC) structure with a lattice spacing of 6.5 Å in a cubic simulation box with length 65.0 Å.

3.2 Composite Polymer Growth

For illustration of the methodology described in the next few sections, linear bead-spring polymer chains are inserted between crystal grains in a polycrystalline structure. The examples in Section A.3 of the Appendix use the polycrystalline sample described in the previous section, where 80 polymer chains are built with an average chain length of 50 monomer units.

3.2.1 Rigid Crystal Grain Expansion

A rigid affine expansion of crystal grains must first be performed to create unoccupied space for which polymer chains can be grown. Individual grains are given unique molecule IDs, and are treated as rigid bodies using the LAMMPS "fix rigid" command. Simulation box expansion is performed using the LAMMPS "fix deform" command, with particle positions remapped accordingly with changes in the box dimensions. The affinely expanded structure is shown in Fig. 2 (right).

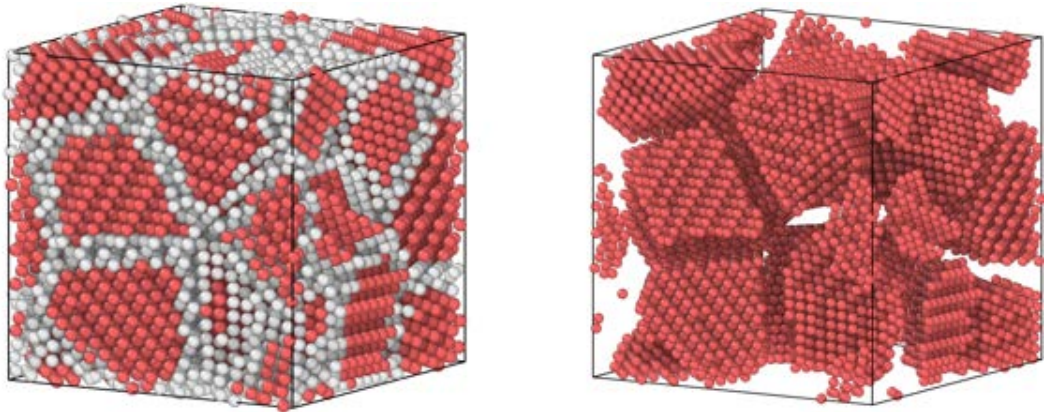

Fig. 2 (Left) Polycrystalline sample of FCC crystal grains output from a Voronoi-based crystal builder program.[7] Boundaries between grains are colored white for visual clarity. (Right) Structure after rigid affine expansion to create space between grains.

3.2.2 Polymer Growth Model

In principle, any particle interaction model can be used during polymer growth. However, in practice the choice of the polymer–polymer interactions significantly influences the resulting polymer chain structure. In this example, polymer–polymer pairwise interactions were turned off during polymer growth or "polymerization" to bias the growth of chains towards elongated structures and minimize self-entanglement. Including a pairwise interaction enhanced chain agglomeration and resulted in aggregates of chains that inhibited crystal grains from approaching to a realistic distance during final compression of the composite. The inclusion of a crystal–polymer interaction is crucial to ensure that polymer particles do not penetrate crystal grains during the polymer growth process. For the example presented in this report, a soft repulsive potential using the LAMMPS "pair_style dpd" command is used for the crystal–polymer interaction, but in principle any interaction potential can be applied. During the polymerization stage, crystal particles are held in place by a harmonic spring force, tethering the particles to their initial position using the LAMMPS "fix spring/self" command.

3.2.3 Monomer Insertion

Following the rigid expansion of the crystal grains, the volume between these grains is filled with monomer particles using the LAMMPS "fix deposit" command, which inserts a particle into the simulation box only if the particle does not overlap (within a specified cutoff distance) any existing particle. In order to enforce chain linearity and eliminate the possibility of forming closed loops, 3 different particle types are used. This methodology could be adapted to produce other chain geometries using additional particle types. All polymer particles

involved in 2 bonds ultimately end up as the first particle type (*polymer type*). The second type (*active type*) represents chain ends that can be involved in a maximum of one bond. The final particle type (*potential type*) represents lone monomer particles that have yet to be incorporated into a polymer chain, and thus are not involved in any bonds.

The insertion of monomer particles is performed in 2 steps. First, *active type* monomer particles are randomly inserted into the volume between crystal grains. Inserting the *active type* particles first, which grow into polymer chains, ensures that these particles are homogeneously distributed throughout the unoccupied volume between grains. This step also provides more precise regulation of the number of chains that will be grown within the sample. Afterwards, *potential type* monomer particles are inserted. The total number of particles that should be inserted can be pre-determined based on the desired mass fraction of the polymer binder in the composite. The number of *active type* particles that are inserted is determined by the desired degree of polymerization. Each initial *active type* monomer grows to become one polymer chain, and therefore the number of *active type* particles should equal the total number of particles divided by the desired degree of polymerization. For example, if 4,000 polymer particles are necessary for a given mass fraction and the desired degree of polymerization is 50, then 80 *active type* monomers and 3,920 *potential type* monomers should be inserted (80 = 4,000/50; 3,920 = 4,000 – 80).

3.2.4 New Bond Creation

Bond creation is implemented using the LAMMPS "fix bond/create" command, where complete details can be found in the LAMMPS documentation. In summary, after every *n* timesteps, bonds are created between particles of specified types if they satisfy a separation distance cutoff criteria. LAMMPS allows various bond topologies (angle, dihedral, improper) when creating new bonds, but is limited to allowing only one new bonding topology type (see LAMMPS documentation for more details).

In the polymer growth algorithm presented here, bonds are only created between chain ends (*active type*) and lone monomers (*potential type*). When a new bond is formed, *potential type* particles become *active type*, and if the new bond was the second bond for the *active type*, the *active type* particle becomes a *polymer type* particle. At the end of the polymerization process, any remaining *potential type* particles are deleted. Remaining *active type* particles are chain end particles. These can either be converted to *polymer type* particles or left as a separate type for distinction later. A schematic representation of the polymerization algorithm is shown in Fig. 3.

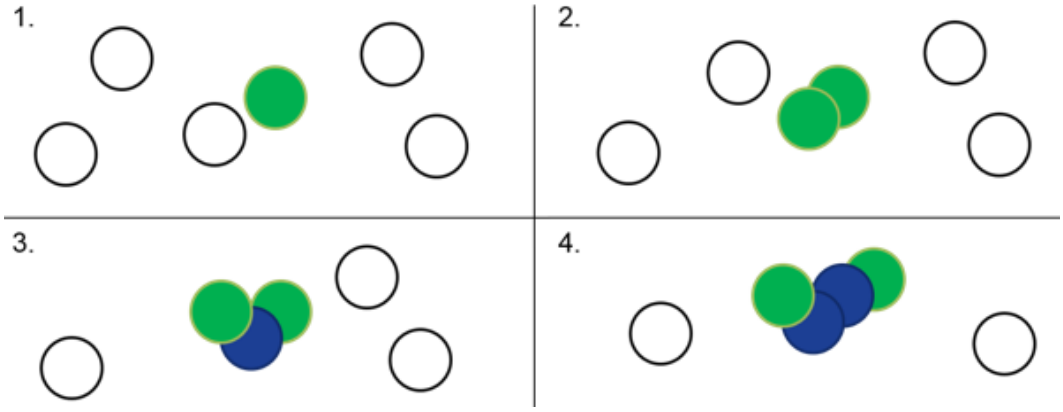

Fig. 3 A schematic showing the polymer growth algorithm. *Potential type* particles are shown in white, *active type* particles are shown in green, and *polymer type* particles are shown in blue. Descriptions for this example: 1) initial system starts with 5 *potential type* particles and 1 *active type* particle; 2) a bond was created between a *potential type* particle and the *active type* particle; 3) another bond has been created, where the middle *active type* particle is converted to a *polymer type* particle; 4) a repeat of the process in Step 3.

The polymerization process does not guarantee 100% complete polymerization, as it depends on the mobility of *potential type* particles and the accessibility of *active type* polymer chain ends. However, if the simulation is run for a sufficiently long time, polymerization can achieve 100% completion. The polymerization process at 0%, 50%, and 100% completion is shown in Fig. 4.

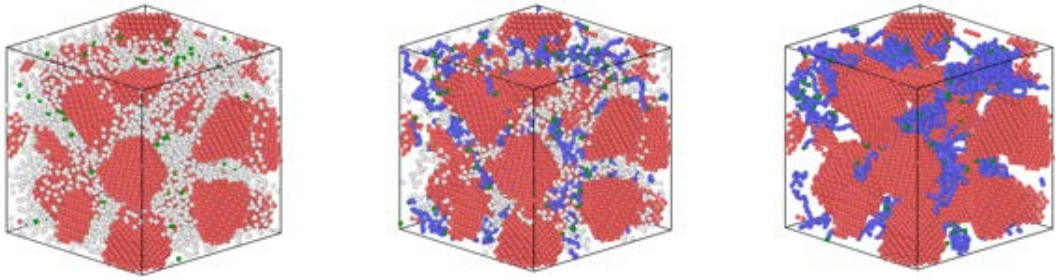

Fig. 4 Polymerization at 0% (left), 50% (middle), and 100% (right) completion. *Active* type particles are colored green, *potential type* particles are colored white, and *polymer type* particles are colored blue.

3.2.5 Compression

Following the polymer growth process, the restraint fixing crystal particles to their positions is removed, polymer–polymer interactions are turned on, and a barostat is applied to control the external pressure on the system. Isothermal–isobaric dynamics are performed to compress the sample to the desired density. The value of the imposed pressure needed to achieve the desired density will be dependent on the specific sample, but convergence of this value

to achieve the desired density within satisfactory limits is expected to occur quickly.

3.3 Updating Molecule IDs

During bond creations using the "fix bond/create" command, LAMMPS does not update molecule IDs; however, this information can be recovered from the bond topology list using graph theory. Using the NetworkX Python package,[8] a graph was built using the bonds listed in the output from LAMMPS. Subgraphs representing different polymer chains can then be extracted from this graph, and a list of nodes (i.e., particles) in each connected subgraph can be used to update the molecule IDs for each polymer chain. The number of nodes in each subgraph is related to the molecular weight of each polymer chain, which is used to define the molecular weight distribution. This algorithm based on graph theory is handled within the NetworkX Python package[8]. This feature was included in the *combat* module, and the following code can be used to generate appropriate molecule IDs:

```
s = combat.System.from_data(filename, atom_style='molecular')
s.molecules_from_bonds()
s.write_data(new_filename, atom_style='molecular')
```

The above code assumes that bonds are present in the LAMMPS data file. Figure 5 (left) shows the average length of the polymer chains as a function of time during the polymerization. Figure 5 (right) shows the final distribution of polymer chain lengths.

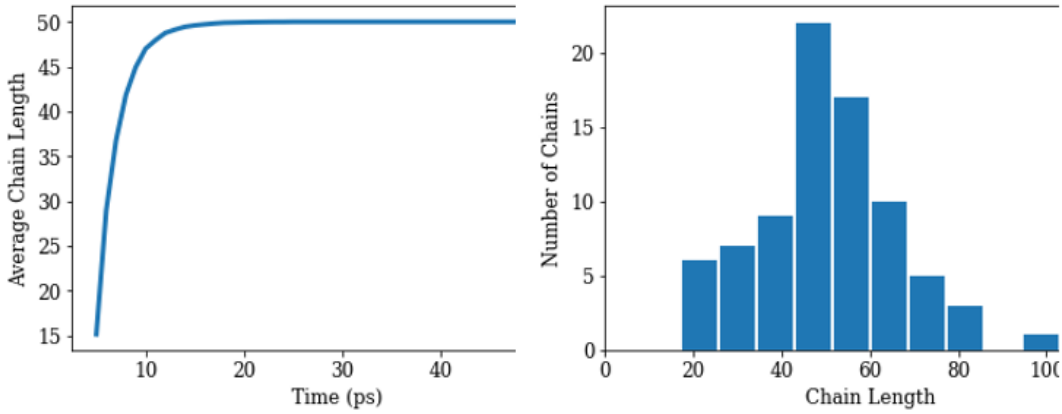

Fig. 5 (Left) Average chain length as a function of simulation time for an example polymerization with a target degree of polymerization of 50. (Right) Final distribution of the polymer chain lengths at the end of the polymerization.

4. Post-Processing Tools

4.1 Void Detection

Voids can form naturally in composite materials and are crucial components of the microstructure heterogeneity in these materials. These voids can have a variety of shapes and sizes, from near-spherical shapes to highly irregular cylindrical shapes, and far beyond. Voids can be introduced during the initial structure construction (intentionally or not), or may form dynamically upon a stimulus (i.e., shock). In either case, it is critical to have the ability to identify the presence and location of voids in such model systems. The algorithm used to probe the model composite structures for unoccupied space involved the construction of a lattice of dummy particles throughout the sample. For each lattice site, an overlap with existing particles (using a user-defined cutoff) results in the deletion of the dummy particle at that lattice site. Any dummy particles remaining after this process effectively fill the unoccupied space in the structure. The construction of the dummy particle lattice and the deletion of the particles overlapping with existing particles were performed using functionality already present in LAMMPS. Additionally within LAMMPS, clusters of dummy particles were identified to distinguish between multiple voids, and the volume of each cluster of dummy particles was calculated using the VORONOI package in LAMMPS, which uses the open-source voro++ package.[9] Example LAMMPS input can be found in Section A.4 of the Appendix. Figure 6 shows the 3 steps during the void detection algorithm: (left) a crystal with a centered spherical void before dummy particle insertion, (middle) the crystal and dummy particles before deletion of overlapping dummy particles, and (right) resulting representation of the void volume within the spherical void.

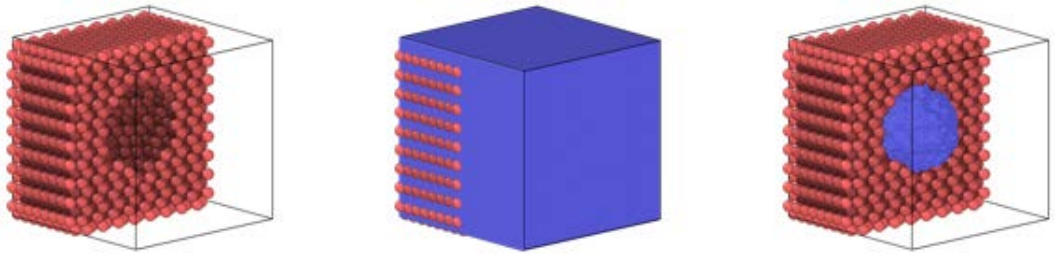

Fig. 6 (Left) A cross-sectional view of a sample structure with a centered spherical void. (Middle) Visualization of both the crystal and a lattice of dummy particles before deletion of overlapping dummy particles. Note that despite the illusion of a continuum-looking blue cube, the image consists of individual blue particles representing dummy particles on a lattice. (Right) Resulting representation of the spherical void following deletion of overlapping dummy particles.

An alternative method for calculating the void volume from the collection of dummy particles is to simply count the number of dummy particles and multiply by the volume per lattice site. This method provides a different physical representation of the void volume, as this is related to the accessible volume of a probe particle. Conversely, the volume resulting from the Voronoi tessellation is related to the void volume, or empty space not occupied by particles' van der Waals spheres. The size of the probe and van der Waals spheres is determined by the cutoff distance used to identify overlaps between system particles and dummy particles.

4.2 Grain Boundary ID

One common defect in polycrystalline materials is the presence of boundaries between crystal grains. These grain boundaries play key roles in the microstructure properties and overall behavior of such materials. The location and identification of grain boundaries can provide a means of probing the specific behaviors associated with these grain boundaries. The algorithm presented here processes LAMMPS dump files under the *combat* Python module and is used to identify and label particles at the grain boundary interfaces and triple junctions. An example script to execute the *combat* Python module is provided in Section A.5 of the Appendix. Note that this algorithm assumes that as input, the LAMMPS dump file contains information that associates each particle with a grain ID (in this case, the molecule ID is used to represent the grain ID, although any column identifying the grain ID can be used generally).

Upon execution of the *combat* module, the neighboring particles within a specified cutoff are identified, as discussed in Section 2.2.3. At this point, each particle is aware of the neighboring particles' positions and grain IDs. For a given particle, the algorithm identifies the number of surrounding grains by searching through the neighbors, identifying the neighbor grain IDs, then counting the number of unique grain IDs, as depicted in Fig. 7. A numerical identifier describing the number of surrounding grains is then assigned to the set of data associated with the current particle. As illustrated in Fig. 8, grain boundary interfaces are formed at the interface between 2 grains (i.e., particles with 2 surrounding grains), whereas triple junction lines are formed at the intersection of 3 crystal grains (i.e., particles with 3 surrounding grains).

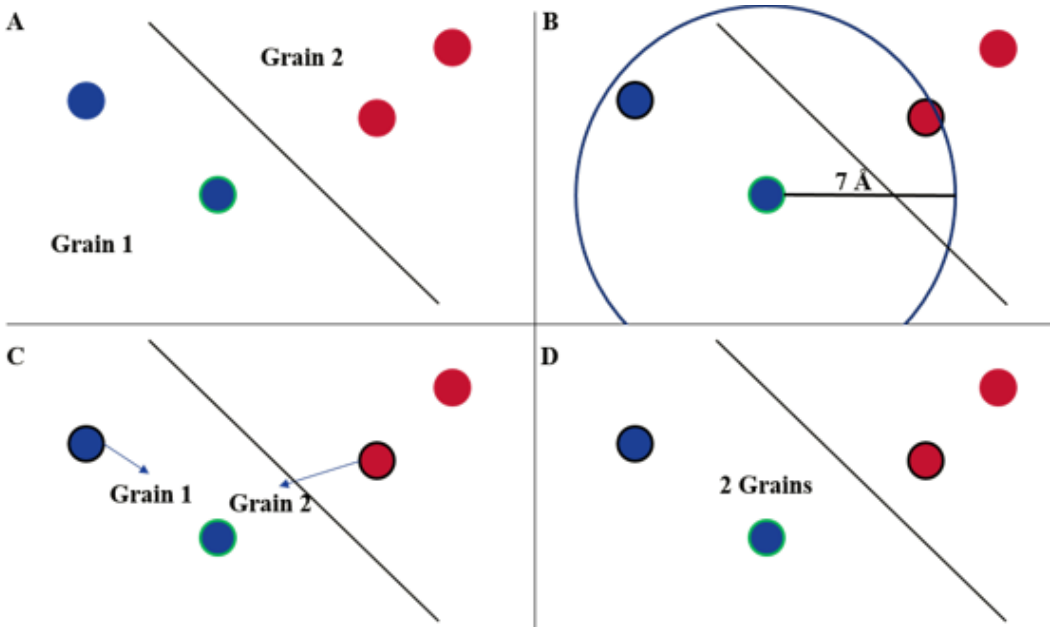

Fig. 7 An illustration of the algorithm used to locate grain boundaries. The processor iterates through the particles, finds the nearest neighbors, and assigns a unique identifier defined by the number of surrounding grains and the type of grains. In this example, the highlighted particle in the center of each frame is located on the boundary between Grains 1 and 2, and receives an identifier accordingly.

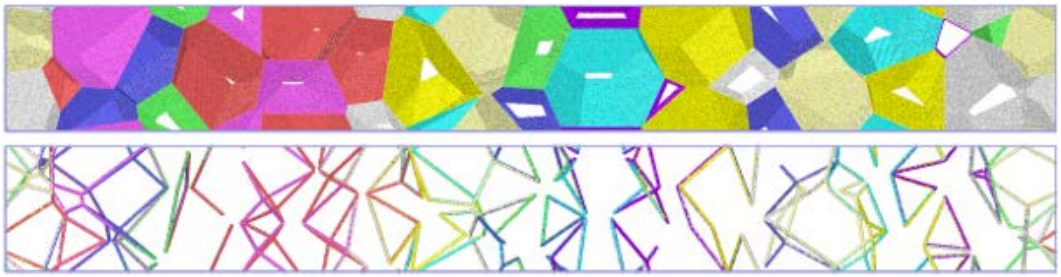

Fig. 8 A 2500 × 300 × 300 nm^3 (1.1-billion particle) polycrystalline sample with an average grain size distribution of 225 nm that has been processed with the *combat* Python module. Grain boundary interfaces (top) and triple junctions (bottom) are shown, where particles are colored by their grain ID.

4.3 Weighted Local Averaging

A per-particle instantaneous property, such as the temperature calculated from instantaneous velocities, can be statistically noisy compared to a global-average property. Therefore, it is of interest to calculate local-average properties for each particle, which includes information from neighboring particles within a cutoff distance. It is common practice to weight the contribution of neighboring particles due to their separation distance from the particle of interest. In general, a local average property (U_i) of particle i can be written as

$$U_i = \frac{\sum_j w(r_{ij}) X_j}{\sum_j w(r_{ij})} \quad (1)$$

where X_j is a per-particle property of particle j, and $w(r_{ij})$ can be any arbitrary weighting function. A common weighting function is the Lucy function,[3] which weights contributions of particles that are closer more than those that are farther away, and has the form

$$w(r_{ij}) = (1 + 3\frac{r_{ij}}{R_c})(1 - \frac{r_{ij}}{R_c})^3 \quad (2)$$

where R_c is the cutoff distance. Figure 9 shows a schematic of the local environment around a particle indicating the contributions from particles within a cutoff distance.

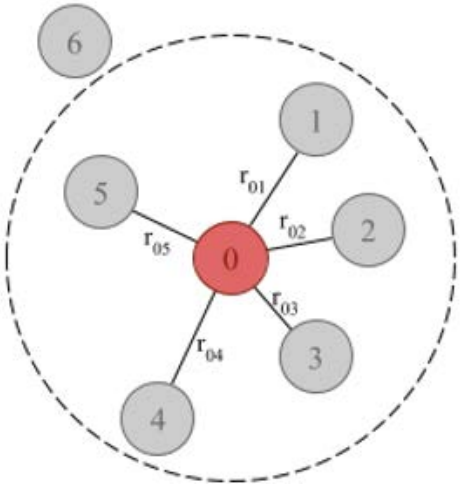

Fig. 9 Local environment around an example particle (red) within a cutoff distance represented by the dashed circle. The local average properties of particle 0 are a function of the properties of Particles 1–5 weighted by their separation distances r. Information from Particle 6 is not included, as it falls outside of the cutoff distance.

Another commonly used weighting function is a quadratic function with the form

$$w(r_{ij}) = A(1 - \frac{r_{ij}}{R_c})^2 \quad (3)$$

where A is an arbitrary pre-factor. Note that to calculate the mean without considering separation distances, set $w(r_{ij}) = 1$.

The *combat* module has been generalized to use any weighting function, and can average any per-particle data present in the DataFrame. The Lucy-weighted local average (implemented by default) can be performed using the following code:

```
s.weighted_avg(data, cutoff=10.0)
```

where `data` is the name of a column in the `s.particles` DataFrame. The following code can be used to implement a new weighting function (e.g., a quadratic function) without any modification to the *combat* module:

```
import numpy as np
def quad(dist):
    ratio = dist/10
       return 15/2/np.pi*np.power(1-ratio, 2)
s.weighted_avg(data, cutoff=10.0, weighting=quad)
```

In this example, `quad` is a user-defined callable function that accepts an array of separation distances, and returns an array of the associated weighting values. The new column in the DataFrame will be assigned a name using the `__name__` attribute of the callable function, for example "weighted_quad_temperature", if a callable function `quad` was used and it averaged a column named "temperature".

A comparison of the instantaneous temperature (calculated from instantaneous velocities) (left), and a Lucy-weighted local average of the temperature (right) is shown in Fig. 10 (example provided in Section A.6 of the Appendix). It is apparent that the local averaging has smoothed out the property of interest, and the local features have become more distinct.

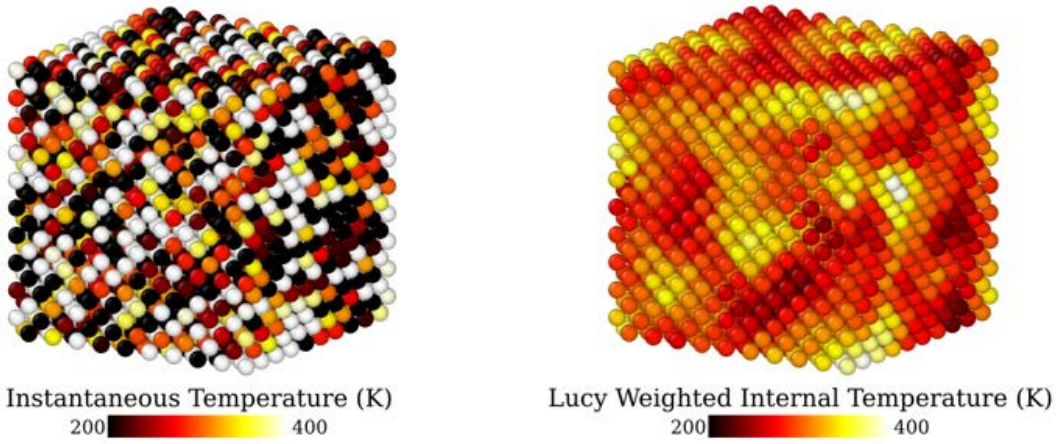

Fig. 10 Comparison of instantaneous temperature (left) and a Lucy-weighted local average temperature (right)

5. Conclusions

The tools presented here enable the efficient construction and characterization of composite models for materials composed of microstructural heterogeneity using functionality present in the LAMMPS software package. Due to the micrometer-size scale of these features, large-scale simulations that generate extremely large data sets of particle properties are required. In order to efficiently

analyze the resulting data, a high-performance Python module was developed implementing MPI and performing efficient vector computation. Although the tools described in this report were originally developed to study composite energetic materials, the tools have been generalized and can be applied to other particle models at any length scale. Additionally, the tools were designed to be flexible to allow weighted local averaging on any per-particle quantity that may be output by LAMMPS, and to allow arbitrary user-defined weighting functions.

6. References

1. Plimpton S. Fast parallel algorithms for short-range molecular dynamics. J Comput Phys.1995;117:1–19.

2. Dalcin L, Paz R, Storti MJ. MPI for Python. Parallel Distrib Comput. 2005;65(9):1108–1115.

3. Lucy LB. A numerical approach to the testing of the fission hypothesis. Astron J. 1977;82:1013.

4. McKinney W. Data structures for statistical computing in Python. Proceedings of the 9th Python in Science Conference. p. 51–56.

5. Jones E, Oliphant T, Peterson P. SciPy: open source scientific tools for Python. SciPy developers; 2001 [accessed 2017 Sep 19]. https://www.scipy.org.

6. van der Walt S, Colbert SC, Varoquaux G. The NumPy array: a structure for efficient numerical computation. CISE. 2011;13 (2);22–30.

7. Foley D, Coleman SP, Tucker G, Tschopp MA. Voronoi-based nanocrystalline generation algorithm for atomistic simulations. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2016 Dec. Report No.: ARL-TN-0806.

8. Hagberg A, Swart PS, Chult D. Exploring network structure, dynamics, and function using NetworkX. Los Alamos (NM): Los Alamos National Laboratory (US); 2008.

9. Rycroft CH. VORO++: a three-dimensional Voronoi cell library in C++. Chaos. 2009;19(4):041111.

Appendix. Examples

The following examples are designed to be executed in their own directory. In some cases, the output produced from one example may be used as input in another example.

A.1 Building a Single Crystal

Table A-1 Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) input for building a face-centered-cubic single crystal

```
log             log.singlecrystal
units           metal
atom_style      atomic
boundary        p p p
lattice         fcc 6.5
region          crystal_box block 0 10 0 10 0 10 units lattice
create_box      1 crystal_box
create_atoms    1 region crystal_box
mass            * 100
velocity        all create 300.0 12345
write_dump      all custom dump.singlecrystal id type x y z vx vy vz
quit
```

A.2. Building a Polycrystalline Sample

Table A-2 Input for nanocrystal_builder.py script

```
single
../singlecrystal/dump.singlecrystal
50
0, 100, 0, 100, 0, 100
none
polycrystal
```

Table A-3 A Python script using the *combat* module to update molecule information

```
import combat
s = combat.System.from_data('data.polycrystal', atom_style='atomic')
s.particles['mol'] = s.particles['type']
s.particles['type'] = 1
s.write_dump('dump.polycrystal')
s.write_data('data.polycrystal.molecules', atom_style='molecular')
```

A.3 Growing Polymer Chains

Table A-4 LAMMPS input for composite polymer growth

```
variable        nchains equal 80
variable        total_particles equal 4000-${nchains}

log             log.grow_polymer
units           metal
atom_style      molecular
boundary        p p p
comm_modify     vel yes

region          box block 0 1 0 1 0 1 units box
create_box      4 box bond/types 1 angle/types 1 &
                extra/bond/per/atom 4 extra/angle/per/atom 9 &
                extra/dihedral/per/atom 18 extra/special/per/atom 27
read_data       ../polycrystal/data.polycrystal.molecules add merge
nocoeff

mass            1 200
mass            2 28.0
mass            3 28.0
mass            4 28.0
pair_style      dpd 1000 12.0 12345
bond_style      harmonic
angle_style     harmonic
special_bonds   lj 0 1 1
pair_coeff      1     1   0     0
pair_coeff      2*4   2*4 0     0
pair_coeff      1     2*4 0.1   1.0
bond_coeff      1 5.0 2.5
angle_coeff     1 4.5 140.00

fix             rigid all rigid molecule torque * off off off &
                force * off off off
variable        expandx equal xhi*1.5
variable        expandy equal yhi*1.5
variable        expandz equal zhi*1.5
fix             deform all deform 1 x final 0.0 ${expandx} &
                y final 0.0 ${expandy} z final 0.0 ${expandz} remap x
dump            particles_dump all custom 10 expansion.dump.* &
                id type mol x y z
run             100

unfix           rigid
unfix           deform
undump          particles_dump

group           xstal type 1
fix             tether xstal spring/self 100.0
region          polymer_growth block EDGE EDGE EDGE EDGE EDGE EDGE &
                units box
thermo          100
thermo_style    custom step temp atoms bonds angles
dump            d_deposit all custom 5000 dump.deposit id type x y z
fix             active all deposit ${nchains} 3 1 40560 region &
                polymer_growth near 6 attempt 1000

unfix           active
```

Table A-4 LAMMPS input for composite polymer growth (continued)

```
fix             potential all deposit ${total_particles} 4 1 27664 &
                region polymer_growth near 6 attempt 1000
run             ${total_particles}

write_data      data.deposit

unfix           potential
undump          d_deposit
group           polymer type 2 3 4
velocity        polymer create 1000.0 82882
timestep        0.005
fix             random_walk all bond/create 1 4 3 5.0 1 atype 1 &
                iparam 1 3 jparam 2 2
fix             nve all nve/limit 0.1
compute         nbonds all property/atom nbonds
dump            particles_dump all custom 100 particles.dump.* &
                id type c_nbonds x y z vx vy vz
compute         bonds polymer property/local batom1 batom2
dump            bonds_dump polymer local 1000 bonds.dump.* &
                c_bonds[1] c_bonds[2]

run             50000

group           unreacted type 4
delete_atoms    group unreacted
group           active type 3
set             group active type 2
write_data      data.expanded_composite nocoeff
quit
```

Table A-5 A Python script using the *combat* module to update molecules from bond topology

```
import combat
s = combat.System.from_data('data.expanded_composite', \
                            atom_style='molecular')
s.molecules_from_bonds()
s.write_data('data.expanded_composite.molecules', \
             atom_style='molecular')
```

A.4. Void Detection

Table A-6 LAMMPS input for void detection

```
log             log.void_detection
units           metal
atom_style      atomic
boundary        p p p
lattice         fcc 6.5
region          crystal_box block 0 15 0 15 0 15 units lattice
create_box      2 crystal_box
create_atoms    1 region crystal_box
group           crystal type 1
region          sph_void1 sphere 30 30 30 20 units box
delete_atoms    region sph_void1
region          sph_void2 sphere 70 70 70 20 units box
delete_atoms    region sph_void2
pair_style      zero 6.5
mass            *         1
pair_coeff      * *

neighbor        0.0 bin

lattice         sc 1
create_atoms    2 box

group           dummy type 2

delete_atoms    overlap 6.5 dummy crystal

compute         voronoi dummy voronoi/atom
compute         voronoi_vol dummy reduce sum c_voronoi[1]

variable        dummy_x atom x
variable        dummy_y atom y
variable        dummy_z atom z

compute         void_cluster dummy cluster/atom 6.5
compute         void_chunk dummy chunk/atom c_void_cluster compress yes
fix             chunk_vol dummy ave/chunk 1 1 1 void_chunk &
                v_dummy_x v_dummy_y v_dummy_z c_voronoi[1] &
                file void.chunk.vol

variable        voronoi_vol equal c_voronoi_vol

dump            void_cluster dummy custom 1 dump.void.dummy &
                id type x y z c_void_chunk c_voronoi[1]
dump            crystal crystal custom 1 dump.void.crystal &
                id type x y z
fix             void_props all print 1 "$(v_voronoi_vol)" &
                append dump.void.props

run 1

quit
```

A.5 Grain Boundary Identification

Table A-7 A Python script using the *combat* module to identify grain boundary particles

```
import combat

# uncomment for parallel version
# from mpi4py import MPI
# comm = MPI.COMM_WORLD

# uncomment for parallel version
# s=combat.System.from_dump('dump.polycrystal',comm=comm, cutoff=10.0)
s = combat.System.from_dump('dump.polycrystal', cutoff=10.0)
s.num_grains = len(set(s.particles['mol'].values))
s.make_neigh_tree()
s.define_grain_boundaries()
s.shrink_boundary_identifiers()
s.particles = s.particles[s.particles['num_grains'] > 1]
del s.particles['neighbors']
# uncomment for parallel version
'''if comm.rank==0:
      print('gathering global number of particles')
nparticles=comm.gather(len(s.particles[s.particles['ghost']==0]), \
                       root=0)
if comm.rank == 0:
      s.global_particles = np.sum(nparticles)
      s.dump_header('dump.header')
s.write_dump('dump.{}p'.format(comm.rank), header=False, ghost=False)
comm.Barrier()
if comm.rank == 0:
      print('concatenating')
      Popen('cat dump.header dump.*p > dump.boundaries, \
            shell=True).communicate()
      Popen('rm dump.header dump.*p', shell=True).communicate()'''
s.write_dump('dump.boundaries')
```

A.6 Weighted Local Averaging Temperature

Table A-8 A Python script using the *combat* module to calculate weight-averaged local temperature

```
import combat
import numpy as np

# uncomment for parallel version
# from mpi4py import MPI
# comm = MPI.COMM_WORLD

# uncomment for parallel version
# s = combat.System.from_dump('dump.singlecrystal', comm=comm,
cutoff=12)
s = combat.System.from_dump('dump.singlecrystal', cutoff=12)

s.make_neigh_tree()

s.temperature(mass=100)
s.weighted_avg('temperature', 12)
def quad(dist):
    ratio = dist/12
    return 15/2/np.pi*np.power(1-ratio, 2)
s.weighted_avg('temperature', 12, weighting=quad)

def avg(dist):
    return [1 for _ in dist]

s.weighted_avg('temperature', 12, weighting=avg)

del s.particles['iloc_neighbors']

# uncomment for parallel version
'''if comm.rank==0:
       print('gathering global number of particles')
nparticles=comm.gather(len(s.particles[s.particles['ghost']==0]), \
                       root=0)
if comm.rank == 0:
       s.global_particles = np.sum(nparticles)
       s.dump_header('dump.header')
s.write_dump('dump.{}p'.format(comm.rank), header=False, ghost=False)
comm.Barrier()
if comm.rank == 0:
       print('concatenating')
       Popen('cat dump.header dump.*p > dump.weighting',
shell=True).communicate()
       Popen('rm dump.header dump.*p', shell=True).communicate()'''

s.write_dump('dump.weighting')
```

List of Symbols, Abbreviations, and Acronyms

3-D	3-dimensional
ARL	US Army Research Laboratory
COMBAT	Composite Model Builder and Analysis Toolkit
FCC	face-centered cubic
I/O	input/output
ID	identification
LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

2 (PDF) DIR ARL
IMAL HRA
RECORDS MGMT
RDRL DCL
TECH LIB

1 (PDF) GOVT PRINTG OFC
A MALHOTRA

19 (PDF) DIR ARL
RDRL WML
N J TRIVEDI
RDRL WML B
D TAYLOR
J P LARENTZOS
J K BRENNAN
B RICE
E F C BYRD
B BARNES
N DANG
J GOTTFRIED
S IZVEKOV
P LAFOND
F DE LUCIA
RDRLWMM F
S COLEMAN
M TSCHOPP
RDRLWMM G
T SIRK
J ANDZELM
R ELDER
B RINDERSPACHER
Y SLIOZBERG

1 (PDF) US NAVAL RSRCH LAB
I SCHWEIGERT

4 (PDF) SANDIA NATIONAL LABS
M WOOD
S J PLIMPTON
A P THOMPSON
S MOORE

1 (PDF) COLORADO SCHOOL OF MINES
G TUCKER

2 (PDF) HPCMP
L DAVIS
C DAHL

1 (PDF) N CAROLINA ST UNIV
M MANSELL

2 (PDF) UNIV OF FLORIDA
DEPT OF CHEM
C COLINA
M FORTUNATO

1 (PDF) PURDUE UNIVERSITY
A STRACHAN

1 (PDF) ENGILITY CORP
T MATTOX

1 (PDF) JE PURKINJE UNIV
M LISAL

1 (PDF) ABERDEEN HIGH SCHOOL
J MATTSON

1 (PDF) UNIVERSITY OF MISSOURI
T SEWELL

1 (PDF) CEA-DAM
JB MAILLET

1 (PDF) LOS ALAMOS NATL LABS
E KOBER

INTENTIONALLY LEFT BLANK.